

# Fast Discovery of Unique and Minimal Data Patterns<sup>1</sup>

KOSTYANTYN DEMCHUK and DOUGLAS J. LEITH, NUI Maynooth, Ireland

A novel fast algorithm for finding quasi identifiers in large datasets is presented. Performance measurements on a broad range of datasets demonstrate reductions in run-time of up to 35 times relative to the state of the art and the scalability of the algorithm to realistically-sized datasets up to several million records.

**Additional Key Words and Phrases:** itemset mining, breadth-first algorithm, frequency-based analysis,  $k$ -anonymity, performance, load balancing.

## ACM Reference Format:

Kostyantyn Demchuk and Douglas J. Leith. 2014. Fast Discovery of Unique and Minimal Data Patterns *ACM V*, N, Article A (January YYYY), 24 pages.  
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

## 1. INTRODUCTION

In this paper we introduce a new algorithm for finding all unique and minimal combinations of  $k$  attributes within a data set. On realistic data sets this algorithm is demonstrated to be up to 35 times faster than the state of the art.

One important application of this algorithm is to find quasi-identifiers in the context of privacy preserving data publishing. A quasi-identifier refers to a subset of attribute values that can uniquely identify one or more entries in a data set and so can potentially be used for re-identification attacks. For example, the seminal study of Sweeney [Sweeney 2002] showed that 87% of the US population are uniquely identified by the three attributes gender, zip code and full date of birth and demonstrated the use of this fact to de-anonymise published health data. With this kind of high-profile attack in mind, due diligence on a new data set prior to publication now commonly requires identifying the combinations of attributes which are unique within the data. This basic first step then allows potential risks to be highlighted and targets for anonymisation identified.

Other potential applications include bioinformatics [Cong et al. 2004; Ordonez et al. 2006], fraud detection [Snchez et al. 2009] and network intrusion [Rahman et al. 2008].

However, searching for unique combinations of attributes is known to be NP-hard [Motwani and Xu 2007], and for realistically sized data sets the long computation times with state of the art algorithms mean that resort may need to be made to approximate sampling approaches with associated increase in risk. It is the need for reasonable processing times while avoiding this risk that originally motivated the development of the algorithm introduced here.

The main contributions of the paper are as follows. Firstly, the introduction of a new algorithm for minimal unique itemset mining, in both sequential and parallel form. Secondly, the derivation of a number of analytic results relating to the properties of minimal unique itemsets – it is these results that underpin the new algorithm and allow the removal of important performance bottlenecks. The basic insight is that by adopting a breadth-first approach it is possible to completely avoid having to scan the (large) dataset in order to carry out support row testing for candidate itemsets. The resulting speedup in execution time comes at the cost of much higher memory usage. However, since available memory size continues to grow year on year while processor speed has largely stagnated in many practical applications this trade-off of memory for speed can be favourable. Thirdly, experimental measurements are presented evaluating the performance of the proposed algorithm on a range of synthetic and application datasets, and comparing this to the popular algorithm MINIT [Haglin and Manning 2007] and the recently proposed algorithm MIWI Miner [Cagliero and P.Garza 2013].

From these tests we find that the proposed algorithm is up to 35 times faster than the state of the art for realistic data sets.

The rest of the paper is organised as follows. In Section 2 we briefly review related work and in Section 3 we introduce notation and some basic definitions. Then, in Section 4 we introduce our new algorithm, explaining its rationale and performance and in Section 5 we present performance measurements for realistic data sets. Finally we summarise our conclusions in Section 6.

## 2. RELATED WORK

Frequent itemset mining has been the subject of extensive study by the data-mining community (*e.g.* see [Agrawal et al. 1996] and the many papers which cite this seminal work). Infrequent itemset mining has attracted less attention, but is of growing interest.

The first algorithm for unique itemset mining (the extreme case of infrequent itemset mining) appears to be SUDA (special unique detection algorithm) proposed in [Elliot et al. 2002]. This was followed shortly afterwards by the development of the SUDA2 algorithm [Manning et al. 2008; Manning and Haglin 2005], which uses a recursive depth-first search approach to generate candidate itemsets from the database of interest (thus every candidate itemset exists in the database) and then efficiently tests these for uniqueness and minimality. Subsequent parallelised implementations are reported in [Yiapanis et al. 2008; Haglin et al. 2009]. SUDA2 lends itself readily to parallelisation by allocating disjoint subtrees to different threads which then carry out a depth-first search on the subtree. However, the work allocated amongst threads may be imbalanced depending on the size and complexity of the subtree assigned to a thread, leading to performance being constrained by the slowest running thread. A number of mitigating strategies are therefore considered in the above papers, and summarised in [Haglin et al. 2009]. SUDA2 is available in the *sdcmicro* package for R [Templ et al. 2013] and is essentially the state-of-the-art algorithm in this area, being used by the UK and Australian national statistics offices [Haglin et al. 2009] and supported by IHSN (International Household Survey Network).

Early work on infrequent (rather than only minimal) itemset mining initially made use of variants of the Apriori algorithm for frequent itemset mining, see [Dong et al. 2007] and references therein, but quickly moved on to algorithms specifically tailored to the infrequent mining task. Almost simultaneously three specialised infrequent itemset algorithms were proposed by [Zhou and Yau 2007], [Szathmary et al. 2007] and [Haglin and Manning 2007]. In [Zhou and Yau 2007] a matrix based scheme referred to as MBS is proposed, involving a direct search of item sequences contained in a database with pruning based on frequency. In [Szathmary et al. 2007] an algorithm referred to as ARIMA is proposed, and later refined in [Szathmary et al. 2012] by the addition of a depth-first search to exclude frequent itemsets. In [Haglin and Manning 2007] the MINIT (minimal infrequent itemsets) algorithm is proposed. MINIT uses a recursive depth-first search with pruning, similarly to the SUDA2 algorithm developed by the same group, and is often used as the baseline algorithm against which the performance of other infrequent mining algorithms is compared. In [Troiano et al. 2009; Troiano and Scibelli 2013] the Rarity algorithm is introduced. Whereas other algorithms start from small itemsets and increase the size as they search, Rarity takes the opposite approach and proceeds from large itemsets to smaller ones (referred to in [Troiano et al. 2009; Troiano and Scibelli 2013] as a top-down strategy). In [Gupta et al. 2011] a pattern-growth recursive depth-first approach is proposed for infrequent itemset mining and two algorithms called IFP\_min and IFP\_MLMS are introduced. In [Gupta et al. 2011] it is observed that there exists a frequency threshold below which MINIT generally outperforms IFP\_min and above which IFP\_min outperforms

MINIT. IFP\_min is also observed to outperform MINIT for large dense datasets. Recently, [Cagliero and P.Garza 2013] extends consideration to the more general task of discovering infrequent weighted itemsets and introduces an algorithm called MIWI Miner. When a weighting of unity is associated with every itemset then this reduces to the infrequent itemset mining problem. For the datasets considered, MIWI Miner is demonstrated to significantly outperform MINIT for infrequent itemset mining apart from the special case of unique itemset mining. However, it is worth noting that the performance comparison here is made only for a small number of datasets since infrequent itemset mining is not the primary focus of the paper.

### 3. PRELIMINARIES

A dataset  $A$  is a table with  $n$  rows and  $m$  columns. The columns in this table contain categorical or finite range continuous data (such as age, income, zip code *etc*). Formally,

**Definition 3.1 (Item).** An item  $a$  is a triple  $(v, j_a, R_a)$  in  $A$ , where  $v \in \mathbb{N}$  is its value,  $j_a \in \{1, \dots, m\}$  is the column of  $A$  containing  $v$ , and  $R_a \subseteq \{1, \dots, n\}$  is the set of  $A$  rows in which the item appears.

Note that we consider items with values from the field of positive integer (natural) numbers  $\mathbb{N}$ , but since any countable set can be mapped on to the integers this restriction is mild (while real values are excluded, finite-precision values are admissible). Letting  $I_A$  denote the set of all items in  $A$  we define uniqueness and uniformity of items in the natural way, as follows:

**Definition 3.2 (Uniqueness).** An item  $a \in I_A$  is unique if  $|R_a| = 1$ . That is, it occurs in dataset  $A$  exactly once. We let  $\delta_A \subseteq I_A$  denote the set of unique items in  $I_A$ .

**Definition 3.3 (Uniformity).** Let  $B \subseteq \{1, \dots, n\}$  be a subset of row indices from dataset  $A$ , and let  $I_B = \{a \in I_A : R_a \cap B \neq \emptyset\}$ . An item  $a$  is said to be uniform in  $I_B$  if  $|R_a \cap B| = |B|$ . That is, item  $a$  occurs in every row of subtable  $B$ . We let  $U_A = \{a \in I_A : |R_a| = n\}$  denote the set of uniform items in  $I_A$ .

**Example 3.4.** For dataset:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 7 & 4 \\ 1 & 6 & 3 & 4 \\ 5 & 2 & 3 & 4 \end{bmatrix}$$

we have:

$$\begin{aligned} I_A &= \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\}), \\ &\quad (4, 4, \{1, 2, 3, 4\}), (5, 1, \{4\}), (6, 2, \{3\}), (7, 3, \{2\})\}. \\ \delta_A &= \{(5, 1, \{4\}), (6, 2, \{3\}), (7, 3, \{2\})\}. \\ U_A &= \{(4, 4, \{1, 2, 3, 4\})\}. \end{aligned}$$

□

**Definition 3.5 (Frequency).** An itemset  $I \subseteq I_A$  is a set of items. A  $k$ -itemset refers to an itemset of cardinality  $k$ . We let  $R_I = \bigcap_{a \in I} R_a$  denote the set of rows in which all items of  $I$  appear, and we refer to  $|R_I|$  as the frequency of itemset  $I$ .

**Definition 3.6 (Unique and Minimal Itemsets).** An itemset  $I^* \subseteq I_A$  is unique and minimal if:

- (1) Uniqueness:  $|R_{I^*}| = 1$ ;

(2) Minimality:  $\forall S \subset I^*, S \neq \emptyset : |R_S| > 1$ .

Note that to establish minimality it is only necessary to test that  $|R_S| > 1$  for sets  $S \subset I^*$  of size  $|I^*| - 1$  since  $R_{S'} \supseteq R_S \forall S' \subset S$ . These  $|I^*| - 1$  subsets are referred to as the *support itemsets* of  $I^*$ . Notice also that itemsets of size 1 (items) are trivially minimal.

We denote the set of all unique and minimal itemsets by  $\mathcal{I}_A^* \subseteq 2^{I_A}$ , where  $2^{I_A}$  denotes the set of all subsets of  $I_A$ . We use calligraphic script to indicate that  $\mathcal{I}_A^*$  is a set of sets and distinguish it from the set of items  $I_A$ .

#### 4. ALGORITHM

In this section we introduce a new algorithm for efficiently finding all of the unique and minimal  $k$ -itemsets up to a user specified size  $k_{max}$ ,  $1 \leq k \leq k_{max} \leq m$ .

##### 4.1. Pre-processing

We begin by observing that uniform items  $u \in U_A$  can be deleted from  $I_A$  as they are not unique and so cannot belong to the set of unique and minimal items  $\mathcal{I}_A^*$  (moreover uniform items cannot form a minimal unique itemset as they break minimality). Further, the set of unique items  $\delta_A$  can be readily identified. The remaining set of non-uniform and non-unique items is  $I'_A := I_A \setminus U_A \setminus \delta_A$  with  $1 < |R_a| < n \forall a \in I'_A$ . The set  $I'_A$  can be partitioned into sets  $L_A$  and  $\bar{L}_A := I'_A \setminus L_A$  such that (i)  $R_a \neq R_b \forall a, b \in L_A$ , (ii)  $\forall c \in \bar{L}_A$  there exists  $d \in L_A$  with  $R_c = R_d$ . That is, within set  $L_A$  no items share the same set of rows. This partitioning can be achieved in the obvious way. Namely, for any set of items in  $I'_A$  which share the same set of rows, add one of these items to  $L_A$  and the rest to  $\bar{L}_A$ . Revisiting Example 3.4, we have  $L_A = \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\})\}$ .

The partitioning into  $L_A$  and  $I'_A \setminus L_A$  possesses the following useful property:

**PROPOSITION 4.1.** *Let  $W \subseteq L_A$  be a minimal unique itemset. Let  $w' \in I_A \setminus L_A$  with  $R_w = R_{w'}$  for some  $w \in W$ . Then  $W \setminus \{w\} \cup \{w'\}$  is also a minimal unique itemset.*

**PROOF.** Since  $W$  is minimal and unique,  $|R_W| = 1$  and  $|R_S| > 1$  for all subsets  $S \subset W$  such that  $|S| = |W| - 1$ ,  $S \neq \emptyset$ . Let  $W' := W \setminus \{w\} \cup \{w'\}$ . We have  $R_{W'} = R_{W \setminus \{w\}} \cap R_{w'} = R_{W \setminus \{w\}} \cap R_w = R_W$  since  $R_w = R_{w'}$ . Hence,  $|R_{W'}| = |R_W| = 1$ . Now consider any subset  $S' \subset W'$  such that  $|S'| = |W'| - 1$ . We have  $|W'| - 1 = |W| - 1$  and either (i)  $S' = S$  when  $w \notin S$  or (ii)  $S' = S \setminus \{w\} \cup \{w'\}$  when  $w \in S$ , where  $S \subset W$ ,  $|S| = |W| - 1$ . Thus, either (i)  $R_{S'} = R_S$  or (ii)  $R_{S'} = R_{S \setminus \{w\}} \cap R_{w'} = R_{S \setminus \{w\}} \cap R_w = R_S$ , respectively. That is,  $|R_{S'}| = |R_S| > 1$  and we are done.  $\square$

It follows that the importance of the partitioning into  $L_A$  and  $I'_A \setminus L_A$  is that after finding the set of unique and minimal itemsets  $\mathcal{L}_A^* \subset 2^{L_A}$  of  $L_A$ , the set of unique and minimal itemsets  $\mathcal{I}_A^* \subset 2^{I_A}$  of  $I_A$  can be obtained immediately. Namely,

**PROPOSITION 4.2.** *For any partition  $(L_A, I'_A \setminus L_A)$  the following holds:  $\mathcal{I}_A^* = \mathcal{L}_A^* \cup \bar{\mathcal{L}}_A^* \cup \delta_A$ , where  $\bar{\mathcal{L}}_A^* := \{I \setminus \{a\} \cup \{b\} : I \in \mathcal{L}_A^*, a \in I, b \in \bar{L}_A, R_a = R_b\}$ .*

**PROOF.** The proposition states that itemset  $I \in \mathcal{I}_A^* \iff I \in \mathcal{L}_A^* \cup \bar{\mathcal{L}}_A^* \cup \delta_A$ . “ $\Leftarrow$ ” If itemset  $I \in \mathcal{L}_A^*$  or  $I \in \delta_A$  then  $I$  is minimal and unique and so  $I \in \mathcal{I}_A^*$ ; if  $I \in \bar{\mathcal{L}}_A^*$  then, by Proposition 4.1,  $I$  is minimal and unique and so  $I \in \mathcal{I}_A^*$ . “ $\Rightarrow$ ” Suppose  $I \in \mathcal{I}_A^*$ . First of all observe that  $\tilde{\mathcal{I}}_A^* = \mathcal{I}_A^*$ , where  $\tilde{I}_A = I_A \setminus U_A$  and  $\tilde{\mathcal{I}}_A^*$  is the set of minimal and unique itemsets in  $2^{\tilde{I}_A}$ . This holds because  $I \cap U_A = \emptyset$  for any  $I \in \mathcal{I}_A^*$  (suppose  $u \in I$ ,  $u \in U_A$  and  $I$  is minimal and unique, then  $R_I = R_{I \setminus \{u\}} \cap R_u = R_{I \setminus \{u\}}$  since  $R_u$  contains all rows of  $A$ ; thus  $|R_{I \setminus \{u\}}| = |R_I| = 1$  what contradicts the minimality of  $I$ ). Further, we

have  $\tilde{\mathcal{I}}_A^* = \hat{\mathcal{I}}_A^* \cup \delta_A$  where  $\hat{I}_A = I_A \setminus U_A \setminus \delta_A$  and  $\hat{\mathcal{I}}_A^*$  is the set of minimal and unique itemsets in  $2^{\hat{I}_A}$ . This is because the elements of  $\delta_A$  are minimal and unique individual items and so if  $I \in \tilde{\mathcal{I}}_A^*$  then either (i)  $I \cap \delta_A = \emptyset$  or (ii)  $|I| = 1$ ,  $I \in \delta_A$  (if  $|I \cap \delta_A| > 1$  then  $|R_I| > |R_{I \cap \delta_A}| = |I \cap \delta_A| > 1$  and so  $I$  is not unique; if  $|I \cap \delta_A| = 1$  and  $|I| > 1$  then  $I$  is not minimal). Hence, we have that  $\mathcal{I}_A^* = \hat{\mathcal{I}}_A^* \cup \delta_A$ . Now  $\hat{I}_A = L_A \cup \bar{L}_A$  with  $L_A \cap \bar{L}_A = \emptyset$ . Hence, if  $I \in \hat{\mathcal{I}}_A^*$  and  $I \cap \bar{L}_A = \emptyset$  (so  $I \subseteq L_A$ ) then  $I \in \mathcal{L}_A$ . If  $I \in \hat{\mathcal{I}}_A^*$  and  $I \cap \bar{L}_A \neq \emptyset$  then  $I \in \bar{\mathcal{L}}_A$  and we are done. Notice that proof works for any partition  $(L_A, I'_A \setminus L_A)$ .  $\square$

In light of Proposition 4.2, our goal can therefore be simplified to finding all unique and minimal  $k$ -itemsets,  $1 \leq k \leq k_{max}$ , of  $L_A$ .

*Example 4.3.* For dataset:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 8 \\ 1 & 2 & 7 & 4 & 8 \\ 1 & 6 & 3 & 4 & 8 \\ 5 & 2 & 3 & 4 & 9 \end{bmatrix}$$

we have:

$$\begin{aligned} I_A &= \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\}), \\ &\quad (4, 4, \{1, 2, 3, 4\}), (5, 1, \{4\}), (6, 2, \{3\}), (7, 3, \{2\}), \\ &\quad (8, 5, \{1, 2, 3\}), (9, 5, \{4\})\}. \\ \delta_A &= \{(5, 1, \{4\}), (6, 2, \{3\}), (7, 3, \{2\}), (9, 5, \{4\})\}. \\ U_A &= \{(4, 4, \{1, 2, 3, 4\})\}. \end{aligned}$$

The remaining set of non-uniform and non-unique items:

$$I'_A := I_A \setminus U_A \setminus \delta_A = \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\}), (8, 5, \{1, 2, 3\})\}.$$

The set  $I'_A$  can be partitioned into sets  $L_A = \{(1, 1, \{1, 2, 3\}), (2, 2, \{1, 2, 4\}), (3, 3, \{1, 3, 4\})\}$  and  $\bar{L}_A := I'_A \setminus L_A = \{(8, 5, \{1, 2, 3\})\}$  such that (i)  $R_a \neq R_b \forall a, b \in L_A$  ( $\{1, 2, 3\} \neq \{1, 2, 4\} \neq \{1, 3, 4\}$  and  $\{1, 2, 3\} \neq \{1, 3, 4\}$ ), (ii)  $\forall c \in \bar{L}_A$  there exists  $d \in L_A$  with  $R_c = R_d$  (for  $(8, 5, \{1, 2, 3\})$  there is  $(1, 1, \{1, 2, 3\})$  in  $L_A$ ).

Denote  $a = (1, 1, \{1, 2, 3\})$ ,  $b = (2, 2, \{1, 2, 4\})$ ,  $c = (3, 3, \{1, 3, 4\})$  and  $d = (8, 5, \{1, 2, 3\})$ . Proposition 4.1 says that if  $\{a, b, c\} \subseteq L_A$  is a minimal unique itemset and  $d \in I_A \setminus L_A$  with  $R_d = R_a$  then  $\{d, b, c\}$  is also a minimal unique itemset. Proposition 4.2 says that for our chosen partition  $(L_A, I'_A \setminus L_A)$  the set of all minimal unique itemsets  $\mathcal{I}_A^*$  can be obtained from the set  $\mathcal{L}_A^* \subset 2^{L_A}$ ,  $\bar{\mathcal{L}}_A^*$  and  $\delta_A$ , in our case:  $\{a, b, c\} \in \mathcal{L}_A^*$ ,  $\{d, b, c\} \in \bar{\mathcal{L}}_A^*$  and unique items from  $\delta_A$ .  $\square$

#### 4.2. Pruning the Search space

Considering the items in  $L_A$  to be an alphabet, all of the possible words in the form of ordered sequences that can be built from  $L_A$  can be represented by a prefix tree. For example, when  $L_A = \{a, b, c, d, e\}$ , the associated prefix tree is shown in the Figure 1. By starting at the root and traversing the branches of the tree, every possible ordered sequence of letters can be obtained.

In principle, the unique and minimal  $k$ -itemsets,  $1 \leq k \leq k_{max}$  of  $L_A$  can be found by traversing every branch of the tree to depth  $k_{max}$  and testing each sequence of items obtained for uniqueness and minimality. However, efficiency can be increased if it is possible to avoid fully traversing every branch i.e. the tree can be pruned.

Basic pruning can be achieved using following fundamental property of itemsets:

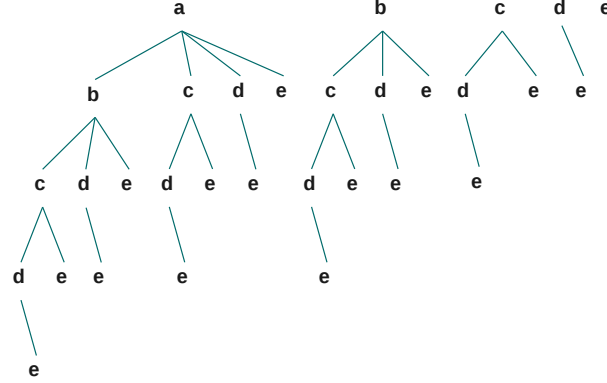


Fig. 1: Prefix tree for the alphabet  $L_A = \{a, b, c, d, e\}$ . By starting at the root and traversing the branches of the tree, every possible ordered sequence of letters can be obtained e.g. traversing the far left-hand branch yields the sequence  $abcde$ .

**PROPOSITION 4.4 (MONOTONICITY).** *Let  $I$  be an itemset. If  $I$  is not minimal then no superset of  $I$  can be minimal.*

**PROOF.** Since  $I$  is non-minimal there exists  $S \subset I, S \neq \emptyset$  such that  $|R_S| \leq 1$ . It follows that  $\forall J \supset I$  there exists  $S \subset J, S \neq \emptyset$  such that  $|R_S| \leq 1$  and so  $J$  is also non-minimal.  $\square$

Hence, as soon as we determine that the sequence of items in an itemset is non-minimal, we can terminate traversal of that branch of the tree. Note that similar pruning is not possible based on uniqueness since a superset of an itemset  $I$  can be unique even if  $I$  is not unique due to the decrease in frequency as more and more items are added to an itemset.

Importantly, the prefix tree associated with itemset  $L_A$  is not unique since the tree depends on how we choose to order the items in  $L_A$ . In general, it is challenging to determine an ordering of items in  $L_A$  which minimises the number of vertices which need to be traversed in the prefix tree in order to find the set  $\mathcal{L}_A^*$  of unique and minimal itemsets of  $L_A$ . We revisit this question later, in Section 5.2.4, but note here that sorting the items of  $L_A$  into ascending order using the following item ordering is efficient for a wide range of datasets.

**Definition 4.5 (Ascending Order).** We order items  $a < b$  if (i)  $|R_a| < |R_b|$  or (ii)  $|R_a| = |R_b|$  and  $j_a < j_b$  or (iii)  $|R_a| = |R_b|, j_a = j_b$  and  $\min R_a < \min R_b$ .

Note that due to the pre-processing and partitioning used to obtain  $L_A$ , for any items  $a \in L_A, b \in L_A \setminus \{a\}$  we must have either  $a < b$  or  $b < a$  i.e. strict total order (if  $j_a = j_b$ ,  $\min R_a = \min R_b$  then items  $a$  and  $b$  are both in the same column  $j_a$  and row  $\min R_a$  of the dataset and so we must have  $a = b$ , but this contradicts the fact that  $b \in L_A \setminus \{a\}$ ). We let  $L_A^<$  denote a list of the items in  $L_A$  sorted in ascending order.

### 4.3. Potential Performance Bottlenecks

To evaluate whether an itemset  $I$  is minimal or not we use the support itemset test from Definition 3.6(2). To evaluate whether an itemset  $I$  is unique, we intersect the

rows of the elements in  $I$  to obtain  $R_I = \cap_{a \in I} R_a$  and test whether  $|R_I| = 1$ . Both of these tests are potentially expensive.

The support itemset test requires enumerating the subsets  $S \subset I$ ,  $|S| = |I| - 1$ , and calculating  $R_S = \cap_{a \in S} R_a$  for each subset. As already noted, testing for uniqueness requires calculating  $R_I = \cap_{a \in I} R_a$ . For large tables, the row sets  $R_a$  may be large and so time consuming to obtain, *e.g.* if the approach taken is to scan the dataset for item  $a$  and record the rows in which  $a$  appears, plus additionally the complexity of calculating  $R_I$  in the obvious manner scales as  $O(|I| \min_{a \in I} |R_a|)$ .

#### 4.4. Kyiv Algorithm

The Kyiv algorithm performs a breadth first search of the prefix tree defined by ordered list  $L_A^<$ . Branches are pruned using Proposition 4.4 – if an itemset  $I$  fails the support itemset test in Definition 3.6(2) then it must be non-minimal and so the subtree with itemset  $I$  at the root can be pruned. The key advantage of the breadth-first approach is that the support row test can be performed extremely efficiently, as discussed in more detail in Section 4.4.1. Pseudo-code for the Kyiv algorithm is given in Algorithm 1.

In the Algorithm 1 the collection of sets  $\{P_\tau\}_{\tau=1}^t$  holds the vertices of level  $k - 1$  of the pruned prefix graph, and the pruned vertices of level  $k$  are stored in  $\{P'_\tau\}_{\tau=1}^{t'}$ . Note that there is never any need to store more than two levels of the pruned prefix graph – we discuss these memory requirements in more detail below. The algorithm visits each vertex in level  $k$  and takes one of three actions: (i) finds the vertex is a non-minimal itemset and so prunes it (it is not added to  $P'$  and its children are not traversed), (ii) find the vertex is a minimal unique itemset and so again prunes it, (iii) finds the vertex is non-unique and its children must be traversed.

In our implementation of Algorithm 1, to hold the prefix graph levels we use a recursive data structure called Graph that stores an array of references to its children of type Graph and other useful data such as current node associated rows that are used for intersection operation. Each child has its identity so fast access to children can be organised via a hashtable which is also stored among the properties of the Graph class.

*Example 4.6.* To illustrate the operation of Algorithm 1, suppose  $k_{max} = 3$  and consider the dataset:

$$A = \begin{bmatrix} * & * & * & 4 & * \\ 1 & 2 & * & 4 & * \\ 1 & 2 & 3 & 4 & * \\ 1 & 2 & 3 & 4 & 5 \\ 1 & * & 3 & * & 5 \\ * & 2 & 3 & * & 5 \\ * & * & * & * & 5 \end{bmatrix}, \text{ where } * - \text{ unique items.}$$

$\delta_A$  is a set of unique items marked by \*. There are no uniform items, so  $U_A = \emptyset$ . There exists single partition of  $I'_A - (L_A, \emptyset)$ , where it can be verified that

$$L_A^< = \{(1, 1, \{2, 3, 4, 5\}), (2, 2, \{2, 3, 4, 6\}), \\ (3, 3, \{3, 4, 5, 6\}), (4, 4, \{1, 2, 3, 4\}), \\ (5, 5, \{4, 5, 6, 7\})\} := \{a, b, c, d, e\}.$$

The prefix tree of  $L_A^<$  is shown schematically in the Figure 1. After line 8 is executed ( $P_1 = \{a\}, P_2 = \{b\}, P_3 = \{c\}, P_4 = \{d\}, P_5 = \{e\}$ ), the first level of the prefix tree is built. The first iteration of the main loop at line 9 (when  $k = 2 < 3 = k_{max}$  and  $t = 5$ ) is reproduced step-by-step below. Here,  $1 \leq i \leq 4 = t - 1, i < j \leq t$  and for each  $(I, J)$  the highest order items are the items contained in  $I$  and  $J$  (which never share a common

**Algorithm 1** Kyiv

---

```

1: Input: dataset  $A$ , threshold  $k_{max}$ 
2: Output: all minimal unique  $k$ -itemsets,  $k \leq k_{max}$ 
3: compute  $I_A = I'_A \cup U_A \cup \delta_A$ 
4: compute  $L_A$  for chosen partition  $(L_A, I'_A \setminus L_A)$ 
5: print unique items in  $\delta_A$  ▷  $k = 1$  case
6: sort  $L_A$  to obtain  $L_A^<$ 
7:  $t \leftarrow 0, k \leftarrow 2$ 
8: foreach  $a \in L_A^<$  do  $t \leftarrow t + 1, P_t \leftarrow \{a\}$ 
9: while  $k \leq k_{max}$  do
10:    $t' \leftarrow 0$ 
11:   foreach  $i \in \{1, \dots, t - 1\}$  do
12:      $I \leftarrow P_i$ 
13:     foreach  $j \in \{i + 1, \dots, t\}$  do
14:        $J \leftarrow P_j$ 
15:       ▷ get the highest order items in  $I$  and  $J$ 
16:        $a \leftarrow \max(I), b \leftarrow \max(J)$ 
17:       if  $I \setminus \{a\} \neq J \setminus \{b\}$  then
18:         break ▷ itemsets do not share a common prefix
19:       ▷ itemsets  $I$  and  $J$  differ exactly by one item now
20:        $W \leftarrow I \cup J$ 
21:       if  $k > 2$  then
22:         ▷ support itemset test, Definition 3.6(2)
23:         if  $\exists S \subset W, |S| = |W| - 1 : |R_S| \leq 1$  then
24:           continue ▷ non-minimal, prune this branch
25:         if  $k = k_{max}$  then
26:           ▷ Lemma 4.7 and Corollary 4.8 (Section 4.4.2)
27:           if  $|R_I| + |R_J| > |R_{I \setminus \{a\}}| + 1$  then continue
28:            $c \leftarrow \max(J \setminus \{b\})$ 
29:           if  $\min(|R_{I \setminus \{c\}}| - |R_I|, |R_{J \setminus \{c\}}| - |R_J|) + 1 < |R_{I \setminus \{c\}} \cap R_b|$  then continue
30:            $R_W \leftarrow R_I \cap R_J$  ▷ intersect rows
31:           if  $|R_W| = 0$  or  $|R_W| = \min(|R_I|, |R_J|)$  then
32:             continue ▷ skip absent and uniform itemsets
33:           if  $|R_W| = 1$  then
34:             print  $W$  ▷ minimal unique itemset found
35:             foreach  $w \in W$  do ▷ apply Proposition 4.1
36:               if  $\exists w' \in I'_A \setminus L_A : R_w = R_{w'}$  then
37:                 print  $W \setminus \{w\} \cup \{w'\}$ 
38:           else ▷ need to store non-unique minimal itemset
39:             if  $k < k_{max}$  then
40:                $t' \leftarrow t' + 1, P_{t'} \leftarrow W$ 
41:           foreach  $t \in \{1, \dots, t'\}$  do  $P_t \leftarrow P_t$ 
42:    $k \leftarrow k + 1, t \leftarrow t'$ 

```

---



prefix). The condition at line 21 is false and there are no absent or uniform itemsets after intersection at line 30:

```

i = 1 : I = P1 = {a}
  j = 2 ≤ 5 = t : J = P2 = {b}, a = max(I), b = max(J), W = {a, b}
    RW = {2, 3, 4}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 1, P'1 = {a, b}
  j = 3 ≤ 5 = t : J = P3 = {c}, a = max(I), c = max(J), W = {a, c}
    RW = {3, 4, 5}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 2, P'2 = {a, c}
  j = 4 ≤ 5 = t : J = P4 = {d}, a = max(I), d = max(J), W = {a, d}
    RW = {2, 3, 4}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 3, P'3 = {a, d}
  j = 5 ≤ 5 = t : J = P5 = {e}, a = max(I), e = max(J), W = {a, e}
    RW = {4, 5}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 4, P'4 = {a, e}
i = 2 : I = P2 = {b}
  j = 3 ≤ 5 = t : J = P3 = {c}, b = max(I), c = max(J), W = {b, c}
    RW = {3, 4, 6}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 5, P'5 = {b, c}
  j = 4 ≤ 5 = t : J = P4 = {d}, b = max(I), d = max(J), W = {b, d}
    RW = {2, 3, 4}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 6, P'6 = {b, d}
  j = 5 ≤ 5 = t : J = P5 = {e}, b = max(I), e = max(J), W = {b, e}
    RW = {4, 6}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 7, P'7 = {b, e}
i = 3 : I = P3 = {c}
  j = 4 ≤ 5 = t : J = P4 = {d}, c = max(I), d = max(J), W = {c, d}
    RW = {3, 4}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 8, P'8 = {c, d}
  j = 5 ≤ 5 = t : J = P5 = {e}, c = max(I), e = max(J), W = {c, e}
    RW = {4, 5, 6}
    4 > |RW| > 1 and k = 2 < 3 = kmax ⇒ t' = 9, P'9 = {c, e}
i = 4 : I = P4 = {d}
  j = 5 ≤ 5 = t : J = P5 = {e}, d = max(I), e = max(J), W = {d, e}
    RW = {4} ⇒ print {d, e} as minimal unique 2-itemset
    Proposition 4.1 is not applied because I'A \ LA = ∅
P1 = {a, b}, P2 = {a, c}, P3 = {a, d}, P4 = {a, e}, P5 = {b, c},
P6 = {b, d}, P7 = {b, e}, P8 = {c, d}, P9 = {c, e}

```

The second iteration of the main loop (when  $k = 3 = k_{max}$  and  $t = 9$ ) is reproduced step-by-step below. Here,  $1 \leq i \leq 8 = t - 1, i < j \leq t$  and for each  $(I, J)$  there were no absent or uniform itemsets after intersection at line 30:

$i = 1 : I = P_1 = \{a, b\}$   
 $j = 2 \leq 9 = t : J = P_2 = \{a, c\}, b = \max(I), c = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, c\}$   
 $k = 3 > 2 :$   
     support itemset test, Definition 3.6(2) at line 23 fails:  
     there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
     Lemma 4.7 (Section 4.4.2) succeeds:  
      $|\{2, 3, 4\}| + |\{3, 4, 5\}| = 6 > 5 = |\{2, 3, 4, 5\}| + 1$   
     Algorithm continues to the next  $J$   
 $j = 3 \leq 9 = t : J = P_3 = \{a, d\}, b = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, d\}$   
 $k = 3 > 2 :$   
     support itemset test, Definition 3.6(2) at line 23 fails:  
     there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
     Lemma 4.7 (Section 4.4.2) succeeds:  
      $|\{2, 3, 4\}| + |\{2, 3, 4\}| = 6 > 5 = |\{2, 3, 4, 5\}| + 1$   
     Algorithm continues to the next  $J$   
 $j = 4 \leq 9 = t : J = P_4 = \{a, e\}, b = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, b, e\}$   
 $k = 3 > 2 :$   
     support itemset test, Definition 3.6(2) at line 23 fails:  
     there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
     Lemma 4.7 (Section 4.4.2) fails:  
      $|\{2, 3, 4\}| + |\{4, 5\}| = 5 \not> 5 = |\{2, 3, 4, 5\}| + 1$   
     Corollary 4.8 (Section 4.4.2) fails:  
      $\min(|\{2, 3, 4, 6\}| - |\{2, 3, 4\}|, |\{4, 5, 6, 7\}| - |\{4, 5\}|) + 1 = 2 \not> 2 = |\{4, 6\}|$   
      $R_W = \{4\} \Rightarrow \text{print } \{a, b, e\} \text{ as minimal unique 3-itemset}$   
     Proposition 4.1 is not applied because  $I'_A \setminus L_A = \emptyset$   
 $j = 5 \leq 9 = t : J = P_5 = \{b, c\}, b = \max(I), c = \max(J)$   
 $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$   
 $i = 2 : I = P_2 = \{a, c\}$   
 $j = 3 \leq 9 = t : J = P_3 = \{a, d\}, c = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, c, d\}$   
 $k = 3 > 2 :$   
     support itemset test, Definition 3.6(2) at line 23 fails:  
     there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
     Lemma 4.7 (Section 4.4.2) succeeds:  
      $|\{3, 4, 5\}| + |\{2, 3, 4\}| = 6 > 5 = |\{2, 3, 4, 5\}| + 1$   
     Algorithm continues to the next  $J$

$j = 4 \leq 9 = t : J = P_4 = \{a, e\}, c = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, c, e\}$   
 $k = 3 > 2 :$   
 support itemset test, Definition 3.6(2) at line 23 fails:  
 there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
 Lemma 4.7 (Section 4.4.2) fails:  
 $|\{3, 4, 5\}| + |\{4, 5\}| = 5 \not\geq 5 = |\{2, 3, 4, 5\}| + 1$   
 Corollary 4.8 (Section 4.4.2) succeeds:  
 $\min(|\{3, 4, 5, 6\}| - |\{3, 4, 5\}|, |\{4, 5, 6, 7\}| - |\{4, 5\}|) + 1 = 2 < 3 = |\{4, 5, 6\}|$   
 Algorithm continues to the next  $J$   
 $j = 5 \leq 9 = t : J = P_5 = \{b, c\}, c = \max(I), c = \max(J)$   
 $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$   
 $i = 3 : I = P_3 = \{a, d\}$   
 $j = 4 \leq 9 = t : J = P_4 = \{a, e\}, d = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{a\} = J \setminus \max(J) : W = \{a, d, e\}$   
 $k = 3 > 2 :$   
 support itemset test, Definition 3.6(2) at line 23 succeeds:  
 $\exists S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$   
 Algorithm continues to the next  $J$   
 $j = 5 \leq 9 = t : J = P_5 = \{b, c\}, d = \max(I), c = \max(J)$   
 $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$   
 $i = 4 : I = P_3 = \{a, e\}$   
 $j = 5 \leq 9 = t : J = P_5 = \{b, c\}, e = \max(I), c = \max(J)$   
 $I \setminus \max(I) = \{a\} \neq \{b\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$   
 $i = 5 : I = P_5 = \{b, c\}$   
 $j = 6 \leq 9 = t : J = P_6 = \{b, d\}, c = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, c, d\}$   
 $k = 3 > 2 :$   
 support itemset test, Definition 3.6(2) at line 23 fails:  
 there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
 Lemma 4.7 (Section 4.4.2) succeeds:  
 $|\{3, 4, 6\}| + |\{2, 3, 4\}| = 6 > 3 = |\{3, 4\}| + 1$   
 Algorithm continues to the next  $J$   
 $j = 7 \leq 9 = t : J = P_7 = \{b, e\}, c = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, c, e\}$   
 $k = 3 > 2 :$   
 support itemset test, Definition 3.6(2) at line 23 fails:  
 there is no  $S \subset W, |S| = |W| - 1 : |R_S| \leq 1$   
 Lemma 4.7 (Section 4.4.2) fails:  
 $|\{3, 4, 6\}| + |\{4, 6\}| = 5 \not\geq 5 = |\{2, 3, 4, 6\}| + 1$   
 Corollary 4.8 (Section 4.4.2) succeeds:

$\min(|\{3, 4, 5, 6\}| - |\{3, 4, 6\}|, |\{4, 5, 6, 7\}| - |\{4, 6\}|) + 1 = 2 < 3 = |\{4, 5, 6\}|$

**Algorithm continues to the next  $J$**

$j = 8 \leq 9 = t : J = P_8 = \{c, d\}, c = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$

$i = 6 : I = P_6 = \{b, d\}$   
 $j = 7 \leq 9 = t : J = P_7 = \{b, e\}, d = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{b\} = J \setminus \max(J) : W = \{b, d, e\}$   
 $k = 3 > 2 :$   
**support itemset test, Definition 3.6(2) at line 23 succeeds:**  
 $\exists S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$   
**Algorithm continues to the next  $J$**

$j = 8 \leq 9 = t : J = P_8 = \{c, d\}, d = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$

$i = 7 : I = P_7 = \{b, e\}$   
 $j = 8 \leq 9 = t : J = P_8 = \{c, d\}, d = \max(I), d = \max(J)$   
 $I \setminus \max(I) = \{b\} \neq \{c\} = J \setminus \max(J) : \text{Algorithm continues to the next } I$

$i = 8 : I = P_8 = \{c, d\}$   
 $j = 9 \leq 9 = t : J = P_9 = \{c, e\}, d = \max(I), e = \max(J)$   
 $I \setminus \max(I) = \{c\} = J \setminus \max(J) : W = \{c, d, e\}$   
 $k = 3 > 2 :$   
**support itemset test, Definition 3.6(2) at line 23 succeeds:**  
 $\exists S = \{d, e\} \subset W, |S| = |W| - 1 : |R_S| = |\{4\}| = 1 \leq 1$   
**Algorithm ends**

To summarise the above, when  $k = 3$  (the ultimate level  $k_{max}$ ), the support itemset test for minimality (line 23), Lemma 4.7 (line 27) and Corollary 4.8 (line 29) are applied in that order to ordered (by the maximum item) pairs of 2-itemsets stored in  $P$  and which share a common prefix. Pairs  $(\{a, d\}, \{a, e\}), (\{b, d\}, \{b, e\}), (\{c, d\}, \{c, e\})$  are pruned by the support itemset test. Pairs  $(\{a, b\}, \{a, c\}), (\{a, b\}, \{a, d\}), (\{a, c\}, \{a, d\}), (\{b, c\}, \{b, d\})$  are pruned by the lemma. Pairs  $(\{a, c\}, \{a, e\}), (\{b, c\}, \{b, e\})$  are pruned by the corollary. Leaving only  $(\{a, b\}, \{a, e\})$  as minimal unique itemsets.  $\square$

**4.4.1. Highly Efficient Support Itemset Testing.** One of the key benefits of adopting a breadth-first approach in Algorithm 1 is that the computational cost of the support itemset test at line 23 can be reduced to essentially zero. This is because the itemsets  $S \subset W$  of size  $|S| = |W| - 1$ , together with the associated row sets  $R_S$ , have already been pre-calculated and stored in data structure  $\mathcal{P} := \{P_\tau\}_{\tau=1}^t$ . Hence, evaluating whether there exists an  $S$  such that  $|R_S| \leq 1$  simply involves lookups from  $\mathcal{P}$ , which can be carried out efficiently using an appropriate data structure for  $\mathcal{P}$  such as a hash table.

Observe that acceleration of the support itemset test at line 23 is achieved in Algorithm 1 at the cost of increased RAM memory to store data structure  $\mathcal{P}$ . This cost is potentially significant, particularly in the middle of the prefix graph where the number of vertices in a level of the graph is largest. However, in view of the fact that the amount of RAM available is growing at a much faster rate than CPU clock speed, this trade-off between increased memory consumption for much reduced computational burden is a favourable one.

**4.4.2. Reducing Number of Row Intersections.** The remaining computational bottleneck of Algorithm 1 is at line 30. We present performance measurements in Section 5 that confirm line 30 accounts for the vast majority of the execution time of Algorithm 1. However, we leave as future work the development of more efficient techniques for computing the intersection operation at line 30.

The potential exists to reduce the number of row intersections at the  $k_{max}$  level of the prefix tree using the following properties:

**LEMMA 4.7.** *Let  $I \subseteq I_A$  be an itemset and  $a, b \in I_A$  any items in  $I_A$ . If*

$$|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + 1 \quad (1)$$

*then  $I \cup \{a, b\}$  is not a unique itemset.*

**PROOF.** We proceed by contradiction. Suppose  $|R_I \cap R_a| + |R_I \cap R_b| > |R_I| + 1$  and itemset  $I \cup \{a, b\}$  is unique (so  $|R_I \cap R_a \cap R_b| = 1$ ). By the distributivity of set intersection,  $R_I \cap (R_a \cup R_b) = (R_I \cap R_a) \cup (R_I \cap R_b)$ . Hence,

$$\begin{aligned} |R_I \cap (R_a \cup R_b)| &= |(R_I \cap R_a) \cup (R_I \cap R_b)| \\ &= |R_I \cap R_a| + |R_I \cap R_b| - |(R_I \cap R_a) \cap (R_I \cap R_b)| \\ &= |R_I \cap R_a| + |R_I \cap R_b| - |R_I \cap R_a \cap R_b| \end{aligned}$$

Now  $|R_I| \geq |R_I \cap (R_a \cup R_b)|$  and by assumption  $|R_I \cap R_a \cap R_b| = 1$ . Hence,  $|R_I| \geq |R_I \cap R_a| + |R_I \cap R_b| - 1$ , yielding the desired contradiction.  $\square$

**COROLLARY 4.8.** *Let  $a_1, \dots, a_k \in I_A$  be any items from  $I_A$ , with  $k > 2$ . If*

$$\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 \quad (2)$$

*then  $\{a_1, \dots, a_k\}$  is not a unique itemset, where*

$$\begin{aligned} \Gamma_0 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_k}| \\ \Gamma_1 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_{k-1}}| \\ \Gamma_2 &:= |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k}| - |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-2}} \cap R_{a_k}| \end{aligned}$$

**PROOF.** There are two cases to consider.

**Case (i):**  $\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 = \Gamma_1 + 1$ . Then

$$\begin{aligned} &|\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_{k-2}}| + |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}} \cap R_{a_k}| \\ &> |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_{k-1}}| + 1 \end{aligned}$$

Let  $I = \cup_{i=1}^{k-3} a_i \cup \{a_{k-1}\}$ ,  $a = a_{k-2}$ ,  $b = a_k$ . By Lemma 4.7  $\{a_1, \dots, a_k\}$  is not unique.

**Case (ii):**  $\Gamma_0 > \min\{\Gamma_1, \Gamma_2\} + 1 = \Gamma_2 + 1$ . Then,

$$\begin{aligned} &|\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k} \cap R_{a_{k-2}}| + |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k} \cap R_{a_{k-1}}| \\ &> |\cap_{i=1}^{k-3} R_{a_i} \cap R_{a_k}| + 1 \end{aligned}$$

Let  $I = \cup_{i=1}^{k-3} a_i \cup \{a_k\}$ ,  $a = a_{k-2}$ ,  $b = a_{k-1}$ . By Lemma 4.7  $\{a_1, \dots, a_k\}$  is not unique.  $\square$

In the final iteration (when  $k = k_{max}$ ) we can use Lemma 4.7 and Corollary 4.8 to test for uniqueness before carrying out the intersection at line 30. If either test concludes that the itemset is non-unique, then there is no need to perform the row intersection.

#### 4.4.3. Correctness

**THEOREM 4.9.** *Algorithm 1 terminates in finite time and finds all minimal unique itemsets of  $I_A$  up to size  $k_{max}$ .*

**PROOF.** Pre-processing from the beginning to the main loop (line 9) is done in finite time: to compute  $I_A$  and  $L_A$  algorithm goes through the  $A$  elements and counts their frequencies while the size of  $A$  is finite ( $n, m < +\infty$ ); printing  $\delta_A$ , sorting  $L_A$  and iterating  $|L_A^<|$  times the loop at line 8 all take finite time as  $|\delta_A|, |L_A| = |L_A^<| < +\infty$ . The search space of the algorithm is the prefix tree which is finite as  $I_A$  is finite. If there is no pruning then Algorithm 1 goes through every branch of maximum length  $k_{max}$  of the tree, otherwise it processes even less number of branches. It takes finite time to process a single branch (that is: navigate it, intersect itemset rows of finite size and either print (Proposition 4.1 takes finite time because  $|W|, |I'_A \setminus L_A| < +\infty$ ) or store the appropriate itemset). Consequently the algorithm terminates in finite time processing all the itemsets of maximum size  $k_{max}$  that have not been thrown out by the support itemset test (line 23), Lemma 4.7 (line 27) and Corollary 4.7 (line 29).

Suppose there is a minimal unique itemset  $I^* \in 2^{I_A}$  that is not found by the algorithm. Proposition 4.2 means that the set of all unique and minimal itemsets  $\mathcal{I}_A^* \subset 2^{I_A}$  can be described by any chosen partition  $(L_A, \bar{L}_A)$ . Thus, either  $I^*$  contains item which does not belong to  $L_A$  or  $|I^*| > k_{max}$ . The former is impossible while the latter does not contradict the theorem.  $\square$

**4.4.4. Parallelisation.** Algorithm 1 can be readily parallelised using shared-memory threads. Namely, at level  $k$  within the prefix tree assign all vertices sharing the same parent at level  $k-1$  within the prefix graph to the same thread and then in each thread execute the loop starting at line 13 in Algorithm 1. The shared memory allows each thread access to the prefix graph information stored in  $P_j, j \in \{i+1, \dots, t\}$ , but there is otherwise no need for inter-thread communication.

When the number of available threads is less than the number of parent vertices at level  $k-1$  in the prefix graph, work must be allocated amongst the threads. As already discussed, the work associated with each parent vertex is dominated by the number of row intersections to be carried out. This number can be accurately estimated based on the number of children of the parent vertex, and so the work associated with each parent vertex estimated in advance. Using these work estimates, load-balanced scheduling of work amongst the threads can then be efficiently realised. As discussed in more detail in Section 5, in this way we can ensure that the running time of all threads is similar thereby enhancing the performance gain from parallelisation – we note that imbalanced thread run times is known to be a key bottleneck in the parallelisation of state-of-the-art depth-first approaches such as SUDA2 and MINIT [Haglin et al. 2009].

**Example 4.10.** Recall Example 4.6. Let  $t = 3$  is the number of threads. When  $k = 2$ , Algorithm 1 allocates jobs between 3 threads: first an empty array  $T$  of size  $t$  is created, each cell of which represents the amount of job to be done by the corresponding thread; then for each item in  $L_A^<$  the number of higher order items is stored in  $T$  at the position of a cell having the minimum value (if there are several such cells, the left-most is chosen). As soon as  $T$  is filled in, all threads start computation immediately. In our case  $T = \{4, 3, 3\}$  and the next job allocation will be accomplished: first thread gets  $\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}$ , second –  $\{b, c\}, \{b, d\}, \{b, e\}$  and the third one –  $\{c, d\}, \{c, e\}, \{d, e\}$ . Row intersection of each ordered pair follows to reveal the unique 2-itemsets and these itemsets are stored in  $P'$ :  $\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, c\}, \{b, d\}, \{b, e\}, \{c, d\}$  and  $\{c, e\}$ ; at the next iteration they will be copied into  $P$  for the  $k = 3$  analysis. Only  $\{d, e\}$  will be printed out as unique and minimal.

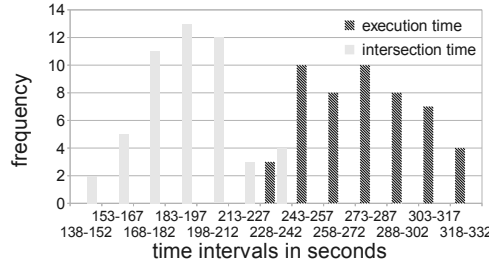


Fig. 2: Distribution of execution and intersection time for randomised datasets,  $k_{max} = 5$ .

When  $k = 3$  (the ultimate level  $k_{max}$ ), job allocation happens as above (now the first level items will have children (2-itemsets) and the job is to intersect their rows to get 3-itemsets). So  $T = \{6, 3, 1\}$  with subsequent job allocation: first thread gets  $(\{a, b\}, \{a, c\})$ ,  $(\{a, b\}, \{a, d\})$ ,  $(\{a, b\}, \{a, e\})$ ,  $(\{a, c\}, \{a, d\})$ ,  $(\{a, c\}, \{a, e\})$ ,  $(\{a, d\}, \{a, e\})$ , second –  $(\{b, c\}, \{b, d\})$ ,  $(\{b, c\}, \{b, e\})$ ,  $(\{b, d\}, \{b, e\})$  and third –  $(\{c, d\}, \{c, e\})$ . As in Example 4.6, the support itemset test, Lemma 4.7 and Corollary 4.8 will eliminate all the pairs inside threads except  $(\{a, b\}, \{a, e\})$ .  $\square$

## 5. EXPERIMENTS

If not otherwise stated, all experiments in this section were held using ascending itemlist order, Lemma 4.7 and Corollary 4.8.

### 5.1. Hardware and Software Setup

We implemented Algorithm 1 in Java (version 1.7.0\_25) using the hppc (version 0.5.2) library, which can be found at <http://labs.carrotsearch.com/hppc.html>. For comparison with the serial version of Algorithm 1, we also implemented a state-of-the-art algorithm MINIT [Haglin and Manning 2007] in Java, using the C++ implementation kindly provided by the developers of MINIT.

For testing we used Amazon cr1.8xlarge instance with an Intel Xeon CPU E5-2670 0 @ 2.60GHz 32 processor (up to 32 hyperthreads), 244Gb of memory, 64-bit Linux operating system (kernel version 3.4.62-53.42. amzn1.x86\_64 of Red Hat 4.6.3-2 Linux distribution (Amazon Linux AMI release 2013.09)).

### 5.2. Domain-Agnostic Performance

**5.2.1. Randomised Datasets.** We begin by investigating performance in a domain-agnostic manner using randomised datasets. Each randomised dataset consisted of 50,000 rows with each row having 25 columns. For each column, the size  $D$  of the domain of element values was selected i.i.d. uniformly at random from the set  $\{10, \dots, 100\}$ . The elements within each column were then selected i.i.d. uniformly at random from domain  $\{1, \dots, D\}$ . On average, for these datasets  $L_A$  contained 1352 items.

**5.2.2. Execution Time.** Figure 2 shows the measured distribution of execution times for Algorithm 1 over 50 randomised datasets when  $k_{max} = 5$ . It can be seen that the execution times are relatively tightly bunched around the mean value of 280 seconds. Also shown in Figure 2 is the corresponding time expended on calculating row intersections at line 30 of Algorithm 1. The mean intersection time is 190 seconds, so 68% of the execution time is expended on row intersections, confirming that these are indeed the primary bottleneck in Algorithm 1. Note that the fraction of execution time expended



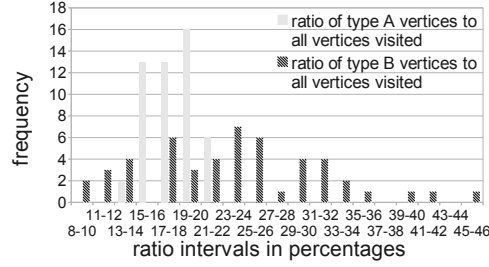


Fig. 3: Distribution of prefix tree vertices traversed for randomised datasets,  $k_{max} = 5$ .

on row intersections depends on  $k_{max}$  and tends to increase as  $k_{max}$  decreases *e.g.* when  $k_{max} = 3$  row intersections absorb 80% of the execution time.

**5.2.3. Prefix tree pruning.** Algorithm 1 carries out online pruning of the prefix tree so as to avoid walking the full prefix tree. Importantly, it also tries to avoid carrying out unnecessary row intersections. We can evaluate the efficiency of the latter by distinguishing between three types of vertices visited: vertices that correspond to minimal unique itemsets (A), vertices which are visited but for which a row intersection is not performed (B) and the rest of the vertices visited (C). Figure 3 shows the distribution of ratios of the number of vertices of types A and B to the total number of prefix tree vertices visited by the algorithm over 50 randomised datasets when  $k_{max} = 5$ . On average 17.5% of vertices visited are type A vertices and 23% type B vertices, although sometimes up to 45% of vertices visited are of type B.

**5.2.4. Impact of Ordering Used for  $L_A$ .** As already noted in Section 4.2, the ordering used to sort set  $L_A$  to obtain  $L_A^<$  can be expected to have an impact on the amount of pruning of the prefix tree achieved, and so on the execution time of Algorithm 1. To investigate this further, we collected performance measurements for three different choices of ordering: (i) ascending order, (ii) descending order (iii) random order (*i.e.* we draw a permutation uniformly at random from the set of permutations mapping from  $\{1, \dots, |L_A|\}$  to itself and apply this permutation to obtain  $L_A^<$ ).

Figure 4 plots the numbers of prefix tree vertices of types A, B and C visited by Algorithm 1 vs the ordering of  $L_A$  used. In this figure data is presented for each of the three orderings (ascending, randomised, descending) and for when Lemma 4.7/Corollary 4.8 are used or not. That is, 6 experiment variants are compared.

It can be seen that use of ascending order significantly reduces the total number of vertices visited, yielding a reduction of roughly a factor of 2 compared to use of a randomised ordering and a factor of 4 compared to descending order. The number of type A vertices visited is, as expected, essentially constant across the tests. However, the number of type B vertices changes significantly and varies such that the number of vertices of type C remains roughly constant. Observe that use of Lemma 4.7 and Corollary 4.8 has little impact on performance in these tests, we will revisit this in Section 5.3.2 where we find that they can speed the runtime up by more than 50%.

Figure 5 plots the corresponding intersection and execution time vs the ordering of  $L_A$  used. It can be seen that the execution time is more sensitive to the ordering than the intersection time. When combined with Figure 4 this allows us to conclude that it is the number of type B vertices that varies strongly with ordering (the number of type A and type C vertices stays nearly constant) and that ascending order reduces execution time primarily by reducing the number of type B vertices *i.e.* by more effective pruning of the search tree which reduces the overall number of vertices visited.

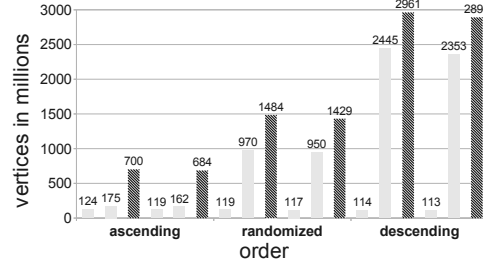


Fig. 4: Prefix tree vertices traversed vs ordering used for  $L_A$ , average over 10 randomised datasets,  $k_{max} = 5$ . For each ordering 6 values are shown: the first three are when Lemma 4.7 and Corollary 4.8 used, the rest – not; in each half the first value represents the number of vertices of type A, second – the number of vertices of type B, third – total number of vertices traversed (that is of type A, B and C).

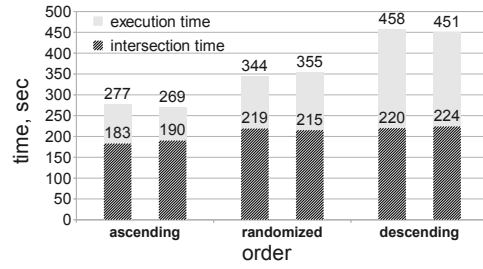


Fig. 5: Intersection and execution time vs ordering used for  $L_A$ , average over 10 randomised datasets,  $k_{max} = 5$  (left bar is when Lemma 4.7/Corollary 4.8 used, right – not).

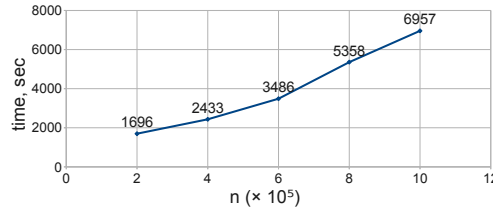


Fig. 6: Execution time vs number of rows  $n$  for a randomised dataset with  $m = 40$  columns,  $k_{max} = 3$ .

**5.2.5. Impact of Dataset Parameters.** To investigate the scaling behaviour of Algorithm 1 to larger datasets we generated a randomised dataset with 1,000,000 rows and 40 columns yielding an itemlist of size 2,179. Taking the first  $n$  rows, Figure 6 plots the execution time of Algorithm 1 versus  $n$  for  $k_{max} = 3$ . It can be seen that the execution time is approximately linear in  $n$ , and so scales well to larger datasets. Although not plotted, memory usage also increased only gradually from 5.6Gb when  $n = 200,000$  to 6Gb when  $n = 1,000,000$ .

Taking the first  $m$  columns of the dataset, Figure 7 plots the execution time versus  $m$  for  $k_{max} = 3$ . It can be seen that the execution time is approximately exponential in

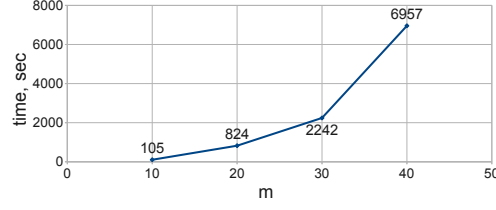


Fig. 7: Execution time vs number of columns  $m$  for a randomised dataset with  $n = 1,000,000$  rows,  $k_{max} = 3$ .

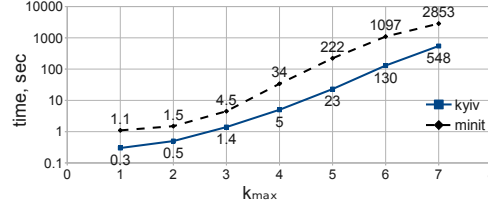


Fig. 8: Execution time vs  $k_{max}$  for connect dataset.

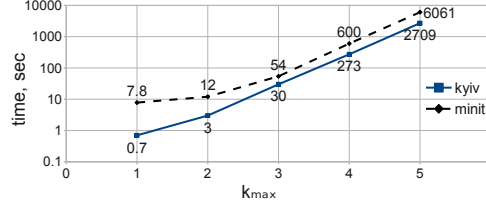
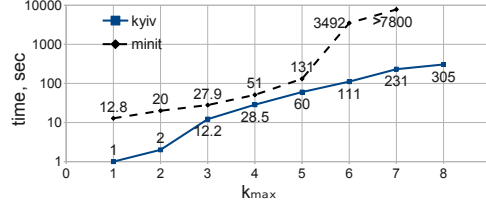
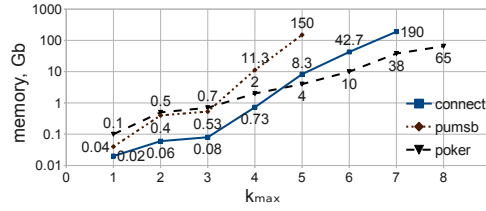
$m$ , and so the algorithm scales less well to datasets with a large number of columns (the size of corresponding itemlist increased from 520 to 2,179). Note that the memory usage also increases quite rapidly with  $m$ , from 0.9Gb when  $m = 10$  to 6Gb when  $m = 40$ .

### 5.3. Domain-Specific Performance

**5.3.1. Datasets.** In this section we collect performance measurements for four domain-specific datasets:

- (1) The Connect-4 dataset is available from <http://fimi.ua.ac.be/data> and contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced. There are 67,557 rows, 43 columns (one for each of the 42 connect-4 squares together with an outcome column - win, draw or lose) and 129 items. It was one of the most computationally challenging datasets to MINIT in [Haglin and Manning 2007].
- (2) The Pumsb dataset is census data for population and housing from the PUMS (Public Use Microdata Sample). This dataset is available from <http://fimi.ua.ac.be/data>. There are 49,046 rows, 74 columns and 1,958 items.
- (3) The Poker dataset is available from <http://archive.ics.uci.edu/ml/datasets.html>. Each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes. There is one Class attribute that describes the "Poker Hand". We removed the last attribute to form a new dataset with 1,000,000 rows, 10 columns and 117 items.
- (4) The USCensus1990 dataset, available from <http://archive.ics.uci.edu/ml/datasets.html>, was collected as part of the 1990 census. We considered subset of this dataset consisting of the first 200,000 rows and 68 columns, which contained 8,009 items.

**5.3.2. Execution Time.** All measurements in the current section are averaged over three consecutive runs of each algorithm.

Fig. 9: Execution time vs  $k_{max}$  for pumsb dataset.Fig. 10: Execution time vs  $k_{max}$  for poker dataset.Fig. 11: Memory consumption of Algorithm 1 vs  $k_{max}$  for three datasets.

Figures 8, 9 and 10 show the measured execution times of Algorithm 1 and MINIT measured for the connect, pumsb and poker datasets vs  $k_{max}$ . It can be seen that Algorithm 1 consistently outperforms MINIT. For the connect dataset it can be seen that Algorithm 1 achieves runtimes between 3 and 9 times faster than MINIT. For the pumsb dataset Algorithm 1 is between 2 and 11 times faster. For the poker dataset Algorithm 1 is between 2 and 33 times faster (for  $k_{max} = 7$  MINIT was terminated after 7,800 seconds without completing).

For the demanding USCensus1990 dataset, which has 8,009 items, both the C++ and Java implementations of MINIT go out of memory and so data is not shown. In comparison, Algorithm 1 finds the minimal sample uniques for  $k_{max} = 4$  in 8 minutes and requiring 22Gb of memory. For  $k_{max} = 3$  the execution time reduces to 2 minutes.

Revisiting the order analysis in Section 5.2.4, we point out that when Algorithm 1 is run without using Lemma 4.7 and Corollary 4.8 then the execution time rises to 269 seconds (from 130 seconds) for the connect dataset,  $k_{max} = 6$  and to 410 (from 273 seconds) seconds for the pumsb dataset,  $k_{max} = 4$  for example.

**5.3.3. Memory Usage.** Algorithm 1 intentionally trades increased memory for faster execution times via its use of a breadth-first approach. This is reasonable in view of the favourable scaling of memory size vs CPU speed on modern hardware. Figure 11 shows the memory consumption of Algorithm 1 for the connect, pumsb and poker datasets vs  $k_{max}$ . These plots indicate the maximum memory needed during algorithm execution

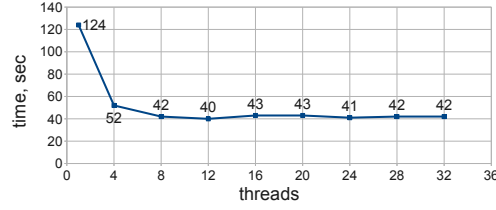


Fig. 12: Parallel algorithm execution time vs threads number for connect,  $k_{max} = 6$ .

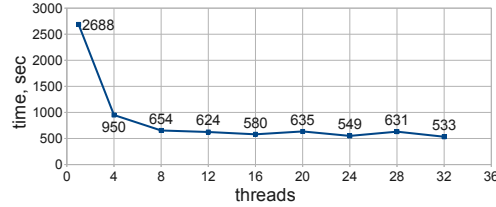


Fig. 13: Parallel algorithm execution time vs threads number for pumsb,  $k_{max} = 5$ .

and so this amount of memory ensures the fastest execution time since garbage collection is not required. For smaller amounts of memory the algorithm is observed to become somewhat slower as the Java Virtual Machine need to start garbage collection.

The memory requirement is dominated by storage of itemset rows to perform intersection. When  $1 < k < k_{max}$ , two levels of the prefix tree must be stored, but when  $k = k_{max}$  (last level), then only one level needs to be stored (for example, the 190Gb in Figure 11 is mostly occupied by the 6-itemset rows). Note that there is a level in the prefix tree that requires the largest amount of memory, a sort of equator. Above this value Algorithm 1 can compute all minimal unique itemsets without additional memory.

#### 5.4. Parallel algorithm performance

Figures 12 and 13 show execution time versus the number of threads used for the connect,  $k_{max} = 6$  and pumsb,  $k_{max} = 5$  datasets respectively. It can be seen that at around 8 threads the performance saturates and additional threads yielding little further performance gain.

In more detail, tables I, II and III show the per thread execution times together with the overall execution time. Data is shown for 4, 8 and 16 threads measured for the pumsb dataset,  $k_{max} = 5$ . It can be seen that the thread execution times consistently have a narrow spread, indicating that the workload is divided evenly amongst the threads. That is, there is not one slow thread which dominates parallel execution time. Observe also that the execution times in the last row of each table (when  $k = 5 = k_{max}$ ) decrease as the number of threads is increased but that the maximum thread execution times when  $k = 3$  and  $k = 4$  do not show a similar decrease. This may be due to the communication overhead when transitioning between layers in the search tree, although we leave detailed analysis of this to future work.

## 6. CONCLUSION

A new algorithm for finding quasi identifiers (unique combinations of  $k$  attributes) within a data set is introduced. This algorithm is demonstrated to be up to 35 times

T	thread 1	thread 2	thread 3	thread 4
$k = 3$				
871	24	24	24	24
$k = 4$				
871	340	344	343	342
$k = 5 = k_{max}$				
871	468	501	470	482

Table I: Granularity of 4 threads for pumsb,  $k_{max} = 5$ . Time is given in seconds, level-wise. T column shows the whole execution time.

T	t1	t2	t3	t4	t5	t6	t7	t8
$k = 3$								
674	21	17	19	21	21	21	19	21
$k = 4$								
674	352	284	354	352	285	291	351	352
$k_{max}$								
674	297	281	293	293	294	282	289	282

Table II: Granularity of 8 threads for pumsb,  $k_{max} = 5$ . Time is given in seconds, levelwise. T column shows the whole execution time.

faster than the state of the art, to scale well to large data sets and to be amenable to parallelisation with well-balanced thread execution times.

## 7. ACKNOWLEDGEMENTS

Our discussions with Aris Gkoulalas-Divanis and the support of Pol Mac Aonghusa and IBM for this work are gratefully acknowledged.

## REFERENCES

- R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. 1996. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining* (1996), 307–328.
- L. Cagliero and P. Garza. 2013. Infrequent weighted itemset mining using frequent pattern growth. *Trans. Knowledge and Data Engineering* (2013).
- G. Cong, A. K. H. Tung, X. Xu, F. Pan, and J. Yang. 2004. Farmer: finding interesting rule groups in microarray datasets. *Proc. SIGMOD* (2004).
- X. Dong, Z. Zheng, Z. Niu, and Q. Jia. 2007. Mining infrequent itemsets based on multiple level minimum supports. *Proc. ICICIC* (2007).
- M. J. Elliot, A. M. Manning, and R. W. Ford. 2002. A computational algorithm for handling the special uniques problem. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems* 10, 5 (2002), 493–509.
- A. Gupta, A. Mittal, and A. Bhattacharya. 2011. Minimally infrequent itemset mining using pattern-growth paradigm and residual trees. *Proc. COMAD* 21 (2011), 1131–1158.
- D. J. Haglin and A. M. Manning. 2007. On minimal infrequent itemset mining. *Proc. Int. Conf. on Data Mining, DMIN* (2007), 141–147.

T	t1	t2	t3	t4	t5	t6	t7	t8
$k = 3$								
567	20	19	19	20	19	19	20	20
$k = 4$								
567	342	345	258	345	342	333	260	346
$k_{max}$								
567	178	171	177	171	170	170	179	179
T	t9	t10	t11	t12	t13	t14	t15	t16
$k = 3$								
567	20	19	19	19	20	19	19	19
$k = 4$								
567	270	272	272	345	271	272	342	345
$k_{max}$								
567	178	177	171	172	172	177	177	177

Table III: Granularity of 16 threads for pumsb,  $k_{max} = 5$ . Time is given in seconds, levelwise. T column shows the whole execution time.

- D. J. Haglin, K. R. Mayes, A. M. Manning, J. Feo, J. R. Gurd, M. Elliot, and J. A. Keane. 2009. Factors affecting the performance of parallel mining of minimal unique itemsets on diverse architectures. *Concurrency and Computation: Practice and Experience* 21, 9 (2009), 1131–1158.
- A. M. Manning, D.J. Haglin, and J. A. Keane. 2008. A recursive search algorithm for statistical disclosure assessment. *Data Mining and Knowledge Discovery* 16, 2 (2008), 165–196.
- A. M. Manning and D. J. Haglin. 2005. A new algorithm for finding minimal sample uniques for use in statistical disclosure assessment. *IEEE International Conference on Data Mining (ICDM05)* (2005), 290–297.
- R. Motwani and Y. Xu. 2007. Efficient algorithms for masking and finding quasi-identifiers. *Proc. VLDB* (2007).
- C. Ordonez, N. Ezquerro, and C. A. Santana. 2006. Constraining and summarizing association rules in medical data. *Knowledge and Information Systems* 9, 3 (2006), 259–283.
- A. Rahman, C. I. Ezeife, and A. K. Aggarwal. 2008. Wi-Fi miner: an online apriori-infrequent based wireless intrusion detection system. *Proc. Sensor-KDD* (2008).
- L. Sweeney. 2002. k-Anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems* 10, 5 (2002), 557–570.
- L. Szathmary, A. Napoli, and P. Valtchev. 2007. Towards rare itemset mining. *Proc. Int. Conf. on Tools with Artificial Intelligence* (2007), 305–312.
- L. Szathmary, P. Valtchev, A. Napoli, and R. Godin. 2012. Efficient vertical mining of minimal rare itemsets. *Proc. Conf. on Concept Lattices and Their Applications* (2012), 269–280.
- D. Snchez, J. M. Serrano, L. Cerda, and M. A. Vila. 2009. Association rules applied to credit card fraud detection. *Expert Systems with Applications* 36, 2 (2009), 3630–3640.
- M. Templ, B. Meindl, and A. Kowarik. 2013. IHSN SDC Introduction. <http://cran.r-project.org/web/packages/sdcMicro/vignettes/guidelines.pdf> (2013).
- L. Troiano and G. Scibelli. 2013. A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets. *Data Mining and Knowledge Discovery* (2013), 1–35.
- L. Troiano, G. Scibelli, and C. Birtolo. 2009. A fast algorithm for mining rare itemsets. *Proc. Int. Conf. on Intelligent Systems Design and Applications* (2009).
- P. Yiapanis, D. J. Haglin, A. M. Manning, K. Mayes, and K. Keane. 2008. Variable-grain and dynamic work generation for minimal unique itemset mining. *Proc. Int. Conf. on Cluster Computing* (2008), 33–41.

- L. Zhou and S. Yau. 2007. Efficient association rule mining among both frequent and infrequent items. *Computers and Mathematics with Applications* 54, 6 (2007), 737–749.